General Information

This page covers the following topics related to the usage of Natural statements:

- User-Defined Variables
- Constants
- Report Specification
- Text Notation
- User Comments
- End of a Statement
- Logical Condition Criteria
- Rules for Arithmetic Assignment
- Renumbering of Source-Code Line Number References

User-Defined Variables

User-defined variables can be used to store intermediate results in a program or routine.

- Naming Conventions
- Definition of Variables
- Statement Reference Notation - r
- Definition of Format and Length
- Special Formats
- Index Notation
- Referencing a Database Array
- Referencing the Internal Count for a Database Array
- Qualifying Data Structures

Naming Conventions

The name of a user-defined variable may be 1 to 32 characters long.

Note:
You may use variable names of over 32 characters (for example, in complex applications where longer meaningful variable names enhance the readability of programs); however, only the first 32 characters are significant and must therefore be unique, the remaining characters will be ignored by Natural.

The name of a user-defined variable must not be a Natural reserved word.

Within one Natural program, you should not use the same name for a user-defined variable and a database field, because this might lead to referencing errors (see Qualifying Data Structures).

The name of a user-defined variable may consist of the following characters:

Character	Explanation
A - Z	alphabetical characters (upper and lower case)
0 - 9	numeric characters
-	hyphen
@	at sign
_	underline
/	slash
$	dollar sign
§	paragraph sign
&	ampersand
#	hash/number sign
+	plus sign (only allowed as first character)

The first character of the name must be one of the following:

- an upper-case alphabetical character
- #
- +
- &

If the first character is a "#", "+" or "&", the name must consist of at least one additional character.

"+" as the first character of a name is only allowed for application-independent variables (AIVs) and variables in a global data area. Names of AIVs must begin with a "+".

"&" as the first character of a name is used in conjunction with dynamic source program modification (see the RUN statement in the Natural Statements documentation), and as a dynamically replaceable character when defining processing rules (see the map editor description in your Natural User's Guide).

Definition of Variables

You define the characteristics of a variable with the following notation:

(r,format-length/index)

This notation follows the variable name, optionally separated by one or more blanks. No blanks are allowed between the individual elements of the notation. The individual elements may be specified selectively as required, but when used together, they must be separated by the characters as indicated above.

Attention:
If operating in structured mode or if a program contains a DEFINE DATA LOCAL clause, variables cannot be defined dynamically in a statement. This does not apply to application-independent variables (AIVs).

Statement Reference Notation - r

A statement label or the source-code line number can be used to refer to a previous Natural statement. This can be used to override Natural's default referencing (as described for each statement, where applicable), or for documentation purposes.

Default Referencing of Database Fields

Generally, the following applies if you specify no statement reference notation: By default, the innermost active database loop (FIND, READ or HISTOGRAM) in which the database field in question has been read is referenced. If the field is not read in any active database loop, the last previous GET statement (in reporting mode also FIND FIRST or FIND UNIQUE statement) which has read the field is referenced.

Referencing with Statement Labels

Any Natural statement which causes a processing loop to be initiated and/or causes data elements to be accessed in the database may be marked with a symbolic label for subsequent referencing.

A label may be specified either in the form label. before the referencing object or in parentheses (label.) after the referencing object (but not both simultaneously).

The naming conventions for labels are identical to those for variables. The period after the label name serves to identify the entry as a label.

Example:

```
... RD. READ PERSON-VIEW BY NAME STARTING FROM 'JONES'   FD. FIND AUTO-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FD.)        DISPLAY NAME (RD.) FIRST-NAME (RD.) MAKE (FD.)
    END-FIND END-READ ...
```

Referencing with Source-Code Line Numbers

A statement may also be referenced by using the number of the source-code line in which the statement is located.

All four digits of the line number must be specified (leading zeros must not be omitted).

Example:

```
... 0110 FIND EMPLOYEES-VIEW WITH NAME = 'SMITH' 0120   FIND VEHICLES-VIEW WITH MODEL = 'FORD' 0130     DISPLAY NAME (0110) MODEL (0120) 0140   END-FIND 0150 END-FIND ...
```

For further information on the referencing of statements, see the Natural Programming Guide.

Definition of Format and Length

Fixed-length variables can be defined with the following formats and corresponding lengths:

Note:
For the definition of Format and Length in dynamic variables, see Definition of Dynamic Variables.

Format		Definable Length (Number of Digits)	Internal Length (in Bytes)
A	Alphanumeric - on mainframe computers: - on all other platforms:	**1 - 253** **1 - 1073741824**	1 - 253 1 - 1073741824
B	Binary - on mainframe computers: - on all other platforms:	**1 - 126** **1 - 1000000000**	1 - 126 1 - 1000000000
C	Attribute Control	-	2
D	Date	-	4
F	Floating Point	**4** or **8**	4 or 8
I	Integer	**1**, **2** or **4**	1, 2 or 4
L	Logical	-	1
N	Numeric (unpacked)	**1 - 29**	1 - 29
P	Packed numeric	**1 - 29**	1 - 15
T	Time	-	7

Length can only be specified if format is specified. With some formats, the length need not be explicitly specified (as shown in the table above).

For fields defined with format N or P, you can use decimal position notation in the form "nn.m". "nn" represents the number of positions before the decimal point, and "m" represents the number of positions after the decimal point. The sum of the values of "nn" and "m" must not exceed 29 and the value of "m" must not exceed 7.

Note:
In reporting mode, if format and length are not specified for a user-defined variable, the default format/length N7 will be used, unless this default assignment has been disabled by the the session parameter FS.

For a database field, the format/length as defined for the field in the DDM apply. (In reporting mode, it is also possible to define in a program a different format/length for a database field.)

In structured mode, format and length may only be specified in a data area definition or with a DEFINE DATA statement.

Example of Format/Length Definition - Structured Mode:

```
DEFINE DATA LOCAL 1 EMPLOY-VIEW VIEW OF EMPLOYEES   2 NAME   2 FIRST-NAME 1 #NEW-SALARY (N6.2) END-DEFINE ... FIND EMPLOY-VIEW ... ... COMPUTE #NEW-SALARY = ... ...
```

In reporting mode, format/length may be defined within the body of the program, if no DEFINE DATA statement is used.

Example of Format/Length Definition - Reporting Mode:


```
...
 FIND EMPLOYEES... ... COMPUTE #NEW-SALARY(N6.2) = ......
```

Special Formats

In addition to the standard alphanumeric (A) and numeric (B, F, I, N, P) formats, Natural supports the special formats C, D, T and L, which are described below.

Format C - Attribute Control

A variable defined with format C may be used to assign attributes dynamically to a field used in a DISPLAY, INPUT or WRITE statement.

For a variable of format C, no length can be specified. The variable is always assigned a length of 2 bytes by Natural.

Example:

```
 DEFINE DATA LOCAL 1 #ATTR(C) 1 #A(N5) END-DEFINE ... MOVE (AD=I CD=RE) TO #ATTR INPUT #A (CV=#ATTR) ...
```

For further information, see the session parameter CV.

Formats D - Date, and T - Time

Variables defined with formats D and T can be used for date and time arithmetic and display. Format D can contain date information only. Format T can contain date and time information; in other words, date information is a subset of time information. Time is counted in tenths of seconds.

For variables of formats D and T, no length can be specified. A variable with format D is always assigned a length of 4 bytes (P6) and a variable with format T is always assigned a length of 7 bytes (P12) by Natural.

Example:

```
DEFINE DATA LOCAL 1 #DAT1 (D)
 END-DEFINE * MOVE *DATX TO #DAT1 ADD 7 TO #DAT1 WRITE '=' #DAT1 END
```

For further information, see the session parameter EM and the system variables *DATX and *TIMX.

The value in a date field must be in the range from 1st January 1582 to 31st December 2699.

Format L - Logical

A variable defined with format L may be used as a logical condition criterion. It can take the value "TRUE" or "FALSE".

For a variable of format L, no length can be specified. A variable of format L is always assigned a length of 1 byte by Natural.

Example:

```
 DEFINE DATA LOCAL 1 #SWITCH(L) END-DEFINE MOVE TRUE TO #SWITCH ... IF #SWITCH    ...    MOVE FALSE TO #SWITCH ELSE    ...    MOVE TRUE TO #SWITCH END-IF
```

For further information on logical value presentation, see the session parameter EM.

Format "Handle"

A variable defined as "HANDLE OF dialog-element-type" can be used as a GUI handle.

A variable defined as "HANDLE OF OBJECT" can be used as an object handle.

For further information on GUI handles, see the Natural User's Guide for Windows. For further information on object handles, see the NaturalX documentation.

Index Notation

An index notation is used for fields that represent an array.

An integer numeric constant or user-defined variable may be used in index notations. A system variable, system function or qualified variable cannot be used in index notations.

Array Definition - Examples:

1. **#ARRAY (3)**
   Defines a one-dimensional array with three occurrences.
2. **FIELD (*label*.,A20/5) or *label*.FIELD(A20/5)**
   Defines an array from a database field referencing the statement marked by "*label*." with format alphanumeric, length 20 and 5 occurrences.
3. **#ARRAY (N7.2/1:5,10:12,1:4)**
   Defines an array with format/length N7.2 and three array dimensions with 5 occurrences in the first, 3 occurrences in the second and 4 occurrences in the third dimension.
4. **FIELD (*label*./i:i + 5) or *label*.FIELD(i:i + 5)**
   Defines an array from a database field referencing the statement marked by "*label*.". FIELD represents a multiple-value field or a field from a periodic group where "*i*" specifies the offset index within the database occurrence. The size of the array within the program is defined as 6 occurrences (i:i + 5). The database offset index is specified as a variable to allow for the positioning of the program array within the occurrences of the multiple-value field or periodic group. For any repositioning of "*i*" a new access must be made to the database via a GET or GET SAME statement.

Natural allows for the definition of arrays where the index does not have to begin with "1". At runtime, Natural checks that index values specified in the reference do not exceed the maximum size of dimensions as specified in the definition.

Note:
For compatibility with Natural Version 1, an array range may be specified using a hyphen (-) instead of a colon (:). A mix of both notations, however, is not permitted. The hyphen notation is only allowed in reporting mode (but not in a DEFINE DATA statement).

On mainframe computers, index values may be in the range from -32767 to +32767. The maximum number of occurrences per array is 32767. The maximum size of an entire array is 32767 bytes (= 32 KB - 1). The maximum size of a data area per programming object is 16,777,215 bytes (16 MB - 1).

On all other platforms, the maximum index value is 1,073,741,824. The maximum size of a data area per programming object is 1,073,741,824 bytes (1 GB). Use the DSLM profile parameter to reduce these limits for compatibility reasons to the limits applicable for mainframe computers.

Simple arithmetic expressions using the "+" and "-" operators may be used in index references. When arithmetic expressions are used as indices, the operators "+" or "-" must be preceded and followed by a blank.

Arrays in group structures are resolved by Natural field by field, not group occurrence by group occurrence.

Example of Group Array Resolution:

```
DEFINE DATA LOCAL 1 #GROUP (1:2)   2 #FIELDA (A5/1:2)   2 #FIELDB (A5) END-DEFINE ...
```

If the group defined above were output in a WRITE statement:

```
WRITE #GROUP (*)
```

the occurrences would be output in the following order:

#FIELDA(1,1) #FIELDA(1,2) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(1) #FIELDB(2)

and `not`:

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDB(1) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(2)
```

Array Referencing - Examples:

1. **#ARRAY (1)**
   References the first occurrence of a one-dimensional array.
2. **#ARRAY (7:12)**
   References the seventh to twelfth occurrence of a one-dimensional array.
3. **#ARRAY (i + 5)**
   References the i+fifth occurrence of a one-dimensional array.
4. **#ARRAY (5,3:7,1:4)**
   Reference is made within a three dimensional array to occurrence 5 in the first dimension, occurrences 3 to 7 (5 occurrences) in the second dimension and 1 to 4 (4 occurrences) in the third dimension.
5. An asterisk may be used to reference all occurrences within a dimension:

```
DEFINE DATA LOCAL
1 #ARRAY1 (N5/1:4,1:4)
1 #ARRAY2 (N5/1:4,1:4)
END-DEFINE
...
ADD #ARRAY1 (2,*) TO #ARRAY2 (4,*)
...
```

Using a Slash before an Array Occurrence

If a variable name is followed by a 4-digit number enclosed in parentheses, Natural interprets this number as a line-number reference to a statement. Therefore a 4-digit array occurrence must be preceded by a slash "/" to indicate that it is an array occurrence; for example:

```
#ARRAY(/1000) not: #ARRAY(1000)
```

because the latter would be interpreted as a reference to source code line 1000.

If an index variable name could be misinterpreted as a format/length specification, a slash "/" must be used to indicate that an index is being specified. If, for example, the occurrence of an array is defined by the value of the variable "N7", the occurrence must be specified as:

```
#ARRAY (/N7) not: #ARRAY (N7)
```


because the latter would be misinterpreted as the definition of a 7-byte numeric field.

Referencing a Database Array

Referencing Multiple-Value Fields and Periodic-Group Fields

A multiple-value field or periodic-group field within a view/DDM may be defined and referenced using various index notations.

For example, the first to tenth values and the Ith to Ith+10 values of the same multiple-value field/periodic-group field of a database record:

```
DEFINE DATA LOCAL 1 I(I2) 1 EMPLOY-VIEW VIEW OF EMPLOYEES   2 LANG (1:10)   2 LANG (I:I + 10) END-DEFINE
```

or:

```
RESET I(I2) ... READ EMPLOYEES OBTAIN LANG(1:10) LANG(I:I + 10)
```

Note:
The same lower bound index may only be used once per array, (this applies to constant indexes as well as variable indexes). For an array definition using a variable index, the lower bound must be specified using the variable by itself, and the upper bound must be specified using the same variable plus a constant.

Referencing Arrays defined with Constants

An array defined with constants may be referenced using either constants or variables. The upper bound of the array cannot be exceeded. The upper bound will be checked by Natural at compilation time if a constant is used.

```
RESET I(I2) I = 1 READ EMPLOYEES OBTAIN LANG(1:10) WRITE LANG(1) / LANG(5:9) / LANG(1:10)


  DEFINE DATA LOCAL 1 I(I2) 1 EMPLOY-VIEW VIEW OF EMPLOYEES   2 LANG (1:10) END-DEFINE *
  READ EMPLOY-VIEW   FOR I 1 TO 5     WRITE LANG(1.I)
    END-FOR END-READ END
```

If a multiple-value field or periodic-group field is defined several times using constants and is to be referenced using variables, the following syntax is used:

```
DEFINE DATA LOCAL 1 I(I2)
 1 J(I2)
 1 EMPLOY-VIEW VIEW OF EMPLOYEES   2 LANG (1:5)   2 LANG (11:20) END-DEFINE *
 READ EMPLOY-VIEW   FOR I 1 TO 2     FOR J I TO 4       DISPLAY 'LANGUAGE' LANG(1.I:J)     END-FOR   END-FOR END-READ END
```

Referencing Arrays defined with Variables

Multiple-value fields or periodic-group fields in arrays defined with variables must be referenced using the same variable.

```
RESET I(I2) I = 1 READ EMPLOYEES OBTAIN LANG(I:I+10) WRITE LANG(I) / LANG (I+5:I+6) END
```

If a different index is to be used, an unambiguous reference to the first encountered definition of the array with variable index must be made. This is done by qualifying the index expression as shown below:

```
RESET I(I2) J(I2) I = 1 J = 1 READ EMPLOYEES OBTAIN LANG(I:I+10) WRITE LANG(I.J) / LANG(I.1:5) END
```

The expression "I." is used to create an unambiguous reference to the array definition and "positions" to the first value within the read array range (LANG(I: I + 10)).

The current content of "I" at the time of the database access determines the starting occurrence of the database array.

Referencing Multiple-Defined Arrays

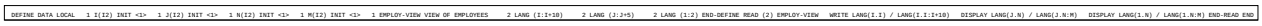
For multiple-defined arrays, a reference with qualification of the index expression is usually necessary to ensure an unambiguous reference to the desired array range.

Example:

```
DEFINE DATA LOCAL   1 I(I2) INIT <1>   1 J(I2) INIT <2>   1 EMPLOY-VIEW VIEW OF EMPLOYEES
      2 LANG (1:10)
      2 LANG (5:20)END-DEFINEREAD (2) EMPLOY-VIEW  WRITE LANG(1.1:10) / LANG(5.5:15)  DISPLAY LANG(1.I:I+2) / LANG(5.J)END-READEND
```

A similar syntax is also used if multiple-value fields or periodic-group fields are defined using index variables.

Example:



Referencing the Internal Count for a Database Array

It is sometimes necessary to reference a multiple-value field and/or a periodic group without knowing how many values/occurrences exist in a given record. Adabas maintains an internal count of the number of values of each multiple-value field and the number of occurrences of each periodic group. This count may be referenced by specifying "C*" immediately before the field name. See also the data-area-editor line command ".*" (as described in your Natural User's Guide).

The count is returned in format N3.

Examples:

C*LANG	Returns the count of the number of values for the multiple-value field LANG.
C*INCOME	Returns the count of the number of occurrences for the periodic group INCOME.
C*BONUS(1)	Returns the count of the number of values for the multiple-value field BONUS in periodic group occurrence 1 (assuming that BONUS is a multiple-value field within a periodic group.)

Note for SQL databases:
The C* notation cannot be used for SQL databases.

Note for VSAM databases:
The C* notation does not return the number of values/occurrences but the maximum occurrence/value as defined in the DDM (MAXOCC).

C* for Multiple-Value Fields Within Periodic Groups

For a multiple-value field within a periodic group, you can also define a C* variable with an index range specification.

The following examples use the multiple-value field BONUS, which is part of the periodic group INCOME. All three examples yield the same result.

Example 1 - Reporting Mode:

```
READ EMPLOYEES BY PERSONNEL-ID FROM 11100117     OBTAIN C*BONUS (1:3)          BONUS (1:3,1:3) *     DISPLAY C*BONUS (1:3)            BONUS (1:3,1:3) END
```

Example 2 - Structured Mode:

```
DEFINE DATA LOCAL 1 EMP VIEW OF EMPLOYEES  2 PERSONNEL-ID  2 INCOME (1:3)  3 C*BONUS  3 BONUS (1:3) END-DEFINE READ EMP BY PERSONNEL-ID FROM 11100117  DISPLAY C*BONUS (1:3)  BONUS (1:3,1:3) END-READ END
```

Example 3 - Structured Mode:

```
DEFINE DATA LOCAL 1 EMP VIEW OF EMPLOYEES  2 PERSONNEL-ID  2 C*BONUS (1:3)  2 INCOME (1:3)  3 BONUS (1:3) END-DEFINE READ EMP BY PERSONNEL-ID FROM 11100117  DISPLAY PERSONNEL-ID C*BONUS (*) BONUS (*,*) END-READ END
```

Note:
As the Adabas format buffer does not permit ranges for count fields, they are generated as individual fields; therefore a C* index range for a large array may cause an Adabas format buffer overflow.

Qualifying Data Structures

To identify a field when referencing it, you may qualify the field; that is, before the field name, you specify the name of the level-1 data element in which the field is located and a period.

If a field cannot be identified uniquely by its name (for example, if the same field name is used in multiple groups/views), you must qualify the field when you reference it.

The combination of level-1 data element and field name must be unique.

Example:

```
DEFINE DATA LOCAL 1 FULL-NAME  2 LAST-NAME (A20)  2 FIRST-NAME (A15) 1 OUTPUT-NAME  2 LAST-NAME (A20)  2 FIRST-NAME (A15) END-DEFINE ... MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME ...
```

The qualifier must be a level-1 data element.

Example:

```
DEFINE DATA LOCAL 1 GROUP1  2 SUB-GROUP  3 FIELD1 (A15)  3 FIELD2 (A15) END-DEFINE ... MOVE 'ABC' TO GROUP1.FIELD1 ...
```

Note:
If you use the same name for a user-defined variable and a database field (which you should not do anyway), you must qualify the database field when you want to reference it; because if you do not, the user-defined variable will be referenced instead.

Constants

- Numeric Constants
- Alphanumeric Constants
- Date and Time Constants
- Hexadecimal Constants
- Logical Constants
- Floating Point Constants
- Handle Constants

Constants are used throughout Natural programs. This section discusses the types of constants that are supported and how they are used.

Numeric Constants

A numeric constant may contain 1 to 29 numeric digits. A numeric constant used in a COMPUTE, MOVE, or arithmetic statement may contain a decimal point and sign notation.

Examples:

```
MOVE 3 TO #XYZ COMPUTE #PRICE = 23.34 COMPUTE #XYZ = -103 COMPUTE #A = #B * 6074
```

Note:
Internally, numeric constants without decimal digits are represented in integer form (format I), while numeric constants with decimal digits, as well as numeric constants without decimal digits that are too large to fit into format I, are represented in packed form (format P).
On mainframe computers, numeric constants are represented internally in packed form (format P); exception: if a numeric constant is used in an arithmetic operation in which the other operand is an integer variable (format I), the numeric constant is represented in integer form (format I).

Validation of Numeric Constants

When numeric constants are used within one of the statements MOVE, COMPUTE, or DEFINE DATA with INIT option, Natural checks at compilation time whether a constant value fits into the corresponding field. This avoids runtime errors in situations where such an error condition can already be detected during compilation.

Alphanumeric Constants

An alphanumeric constant may contain 1 to 253 alphanumeric characters.

An alphanumeric constant must be enclosed in either apostrophes (’) or quotation marks (").

Examples:

```
MOVE ’ABC’ TO #FIELDX MOVE ’% INCREASE’ TO #TITLE DISPLAY "LAST-NAME" NAME
```

An alphanumeric constant that is used to assign a value to a user-defined variable must not be split between statement lines.

Apostrophes Within Alphanumeric Constants

If you want an apostrophe to be part of an alphanumeric constant that is enclosed in apostrophes, you must write this as two apostrophes or as a single quotation mark.
If you want an apostrophe to be part of an alphanumeric constant that is enclosed in quotation marks, you write this as a single apostrophe.

Example:

If you want the following to be output:

```
HE
SAID, ’HELLO’
```

you can use any of the following notations:

```
WRITE ’HE SAID, ’’HELLO’’’ WRITE ’HE SAID, "HELLO"’ WRITE "HE SAID,
""HELLO""" WRITE "HE SAID, ’HELLO’"
```


Note:
If quotation marks are not converted to apostrophes as shown above, this is due to the setting of the profile parameter TQ; ask your Natural administrator for details.

Concatenation of Alphanumeric Constants

Alphanumeric constants may be concatenated to form a single value by use of a hyphen.

Examples:

```
MOVE 'XXXXXX' -      'YYYYYY' TO #FIELD  MOVE "ABC" - 'DEF' TO #FIELD
```

In this way, alphanumeric constants can also be concatenated with hexadecimal constants.

Date and Time Constants

A date constant may be used in conjunction with a format D variable. Date constants may have the following formats:

D'YYYY-MM-DD'	International date format
D'DD.MM.YYYY'	German date format
D'DD/MM/YYYY'	European date format
D'MM/DD/YYYY'	USA date format

Example:

```
DEFINE DATA LOCAL 1 #DATE (D) END-DEFINE ... MOVE D'1997-04-27' TO #DATE ...
```

The default date format is controlled by the profile parameter DTFORM as set by the Natural administrator.

A time constant may be used in conjunction with a format T variable. A time constant has the following format:

T'*hh:ii:ss*'

where

Character	Explanation
hh	hours
ii	minutes
ss	seconds

Example:

```
DEFINE DATA LOCAL 1 #TIME (T) END-DEFINE ... MOVE T'11:33:00' TO #TIME ...
```


Extended Time Constants

A time variable (format T) can contain date and time information, date information being a subset of time information; however, with a "normal" time constant (prefix "T") only the time information of a time variable can be handled:

T'*hh:ii:ss*'

With an extended time constant (prefix "E"), it is possible to handle the full content of a time variable, including the date information:

E'*yyyy-mm-dd hh:ii:ss*'

Apart from that, the use of an extended time constant in conjunction with a time variable is the same as for a normal time constant.

Note:
The format in which the date information has to be specified in an extended time constant depends on the setting of the profile parameter DTFORM. The extended time constant shown above assumes DTFORM=I (international date format).

Hexadecimal Constants

A hexadecimal constant may be used to enter a value which cannot be entered as a standard keyboard character.

A hexadecimal constant is prefixed with an "H". The constant itself must be enclosed in apostrophes and may consist of the hexadecimal characters 0 - 9, A - F. Two hexadecimal characters are required to represent one byte of data.

The hexadecimal representation of a character varies depending on whether your computer uses an ASCII or EBCDIC character set. When you transfer hexadecimal constants to another computer, you may therefore have to convert the characters.

```
ASCII Examples:
 H'313233'    (equivalent to the alphanumeric constant '123') H'414243'    (equivalent to the alphanumeric constant 'ABC')
EBCDIC Examples:
 H'F1F2F3'    (equivalent to the alphanumeric constant '123') H'C1C2C3'    (equivalent to the alphanumeric constant 'ABC')
```

Hexadecimal constants may be concatenated by using a hyphen between the constants.

```
ASCII Examples:
 H'414243' - H'444546' (equivalent to 'ABCDEF')
EBCDIC Examples:
 H'C1C2C3' - H'C4C5C6' (equivalent to 'ABCDEF')
```

In this way, hexadecimal constants can also be concatenated with alphanumeric constants.

Note:
When a hexadecimal constant is transferred to another field, it will be treated as an alphanumeric value.
Under UNIX, if a hexadecimal constant is output that contains any characters from the ranges H'00' to H'1F' or H'80' to H'A0', these characters will not be output, as they would be interpreted as terminal control characters. As of version 2.2 these hex constants are not suppressed.

Logical Constants

The logical constants "TRUE" and "FALSE" may be used to assign a logical value to a variable defined with format L.

Example:

```
DEFINE DATA LOCAL 1 #FLAG (L) END-DEFINE ... MOVE TRUE TO #FLAG ... IF #FLAG ...    statement ...    MOVE FALSE TO #FLAG END-IF ...
```

Floating Point Constants

Floating point constants can be used with variables defined with format F.

Example:

```
DEFINE DATA LOCAL 1 #FLT1 (F4) END-DEFINE ... COMPUTE #FLT1 = -5.34E+2 ...
```

See information on arithmetic involving floating-point numbers.

Attribute Constants

Attribute constants can be used with variables defined with C format. This type of constant must be enclosed within parentheses.

The following attributes may be used:

AD=D	default	**CD=BL**	blue
AD=B	blinking	**CD=GR**	green
AD=I	intensified	**CD=NE**	neutral
AD=N	non-display	**CD=PI**	pink
AD=V	reverse video	**CD=RE**	red
AD=U	underlined	**CD=TU**	turquoise
AD=C	cursive/italic	**CD=YE**	yellow
AD=Y	dynamic attribute		
AD=P	protected		

Example:

```
DEFINE DATA LOCAL 1 #ATTR (C) 1 #FIELD (A10) END-DEFINE  ... MOVE (AD=I CD=BL) TO #ATTR ... INPUT #FIELD (CV=#ATTR) ...
```

Handle Constants

The handle constant NULL-HANDLE can be used with GUI handles and object handles.

For further information on GUI handles, see the Natural User’s Guide for Windows. For further information on object handles, see the NaturalX documentation.

Report Specification - *rep*

"*(rep)*" is the output report identifier for which a statement is applicable. If a Natural program is to produce multiple reports, the notation "*(rep)*" must be specified with each output statement which is to be used to create output for any report other than the first report (report 0). A value of 1 - 31 may be specified.

On mainframe computers, this notation only applies to reports created in batch mode, to reports under Com-plete, CMS, IMS/TM or TIAM; or when using Natural Advanced Facilities under CICS, TSO or UTM.

Examples:

```
DISPLAY (1) NAME ...WRITE (4) NAME ...
```

The value for (rep) may also be a logical name which has been assigned using the DEFINE PRINTER statement.

Example:

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'WRITE (LIST) NAME ...
```

Text Notation

'text' specifies text to be used in conjunction with an INPUT, DISPLAY, WRITE, WRITE TITLE or WRITE TRAILER statement. The text must be enclosed in either apostrophes (') or quotation marks ("). The text itself may be 1 to 72 characters and must not be continued from one line to the next. Text elements may be concatenated by using a hyphen.

Examples:

```
REINPUT 'PLEASE ENTER A VALID VALUE' WRITE    "NEW SALARY" #NEW-SALARY WRITE    'TEXT1'-'TEXT2'-'TEXT3'
```

If you want an apostrophe to be part of a text string that is enclosed in apostrophes, you must write this as two apostrophes or as a single quotation mark. Either notation will be output as a single apostrophe.
If you want an apostrophe to be part of a text string that is enclosed in quotation marks, you write this as a single apostrophe.

Examples:

```
 #FIELDA = 'O''CONNOR' #FIELDA = 'O"CONNOR' #FIELDA = "O'CONNOR"

In all three cases, the result will be:

 O'CONNOR
```

Note:
If quotation marks are not converted to apostrophes as shown above, this is due to the setting of the profile parameter TQ; ask your Natural administrator for details.

If a single character is to be output several times as text, you use the following notation:

```
'c'(n)
```

As c you specify the character, and as n the number of times the character is to be generated. The maximum value for n is 249.

Example:

```
WRITE '*'(3)
```

Instead of apostrophes before and after the character c you can also use quotation marks.

User Comments

You have the following possibilities for entering your comments in source code:

- If you wish to use an entire source-code line for a user comment, you enter one of the following at the beginning of the line:
  - an asterisk and a blank (*),
  - two asterisks (**), or
  - a slash and an asterisk (/*):

    ```
    *  USER COMMENT
    ** USER COMMENT
    /* USER COMMENT
    ```

- If you wish to use only the latter part of a source-code line for a user comment, you enter a blank, a slash and an asterisk (/*); the remainder of the line after this notation is thus marked as a comment:

  ```
  ADD 5 TO #A          /* USER COMMENT
  ```

End of a Statement

To explicitly mark the end of a statement, you can place a semicolon (;) between the statement and the next statement. This can be used to make the program structure clearer, but is not required.

Logical Condition Criteria

- Relational Expression
- Extended Relational Expression
- MASK Option
- SCAN Option
- BREAK Option
- IS Option
- Evaluation of a Logical Variable
- Modified Control Variables
- SPECIFIED Option
- Fields Used Within Logical Condition Criteria
- Logical Operators in Complex Logical Expressions

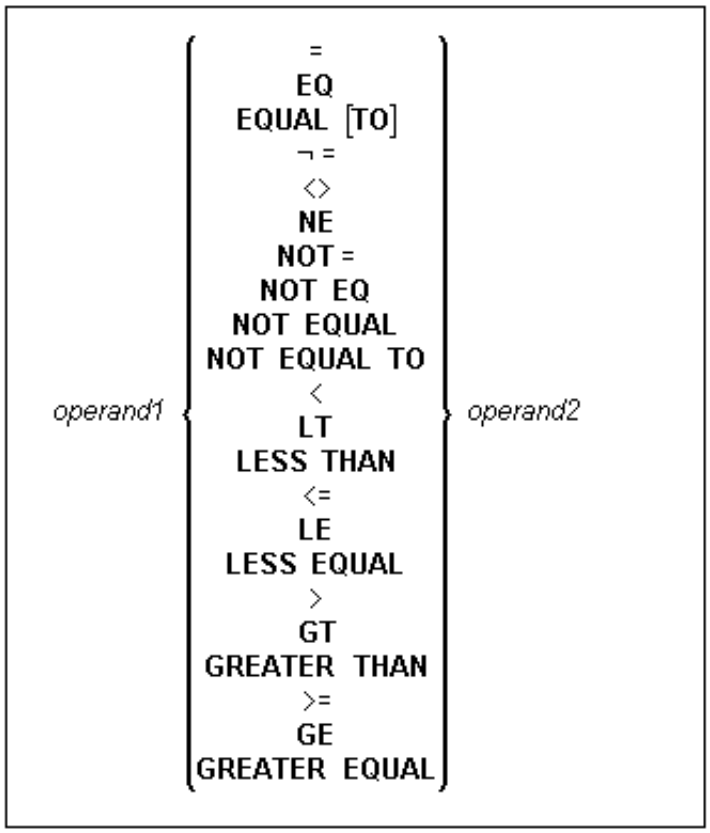
The basic criterion is a relational expression. Multiple relational expressions may be combined with logical operators (AND, OR) to form complex criteria.

Arithmetic expressions may also be used to form a relational expression.

Logical condition criteria can be used in the following statements:

Statement	Usage
FIND	A WHERE clause containing logical condition criteria may be used to indicate criteria in addition to the basic selection criteria as specified in the WITH clause. The logical condition criteria specified with the WHERE clause are evaluated after the record has been selected and read. In a WITH clause, "basic search criteria" (as described with the FIND statement) are used, but not logical condition criteria.
READ	A WHERE clause containing logical condition criteria may be used to specify whether a record that has just been read is to be processed. The logical condition criteria are evaluated after the record has been read.
HISTOGRAM	A WHERE clause containing logical condition criteria may be used to specify whether the value that has just been read is to be processed. The logical condition criteria are evaluated after the value has been read.
ACCEPT/REJECT	An IF clause may be used with an ACCEPT or REJECT statement to specify logical condition criteria in addition to that specified when the record was selected/read with a FIND, READ, or HISTOGRAM statement. The logical condition criteria are evaluated after the record has been read and after record processing has started.
IF	Logical condition criteria are used to control statement execution.
DECIDE FOR	Logical condition criteria are used to control statement execution.
REPEAT	The UNTIL or WHILE clause of a REPEAT statement contain logical condition criteria which determine when a processing loop is to be terminated.

Relational Expression



Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	E	A	N	P	I	F	B	D	T	L		G	O	yes	yes
Operand2	C	S	A		N	E	A	N	P	I	F	B	D	T	L		G	O	yes	no

For an explanation of the Operand Definition Table shown above, see Syntax Symbols and Operand Definition Tables in the Natural Statements documentation. In the "Possible Structure" section of the table above, "E" stands for arithmetic expressions; that is, any arithmetic expression may be specified as an operand within the relational expression.

Examples:

```
IF NAME = 'SMITH' IF LEAVE-DUE GT 40 IF NAME = #NAME
```

For information on comparing arrays in a relational expression, see Processing of Arrays.

Note:
If a floating-point operand is used, comparison is performed in floating point. Floating-point numbers as such have only a limited precision; therefore, rounding/truncation errors cannot be precluded when numbers are converted to/from floating-point representation.

Arithmetic Expressions in Logical Conditions

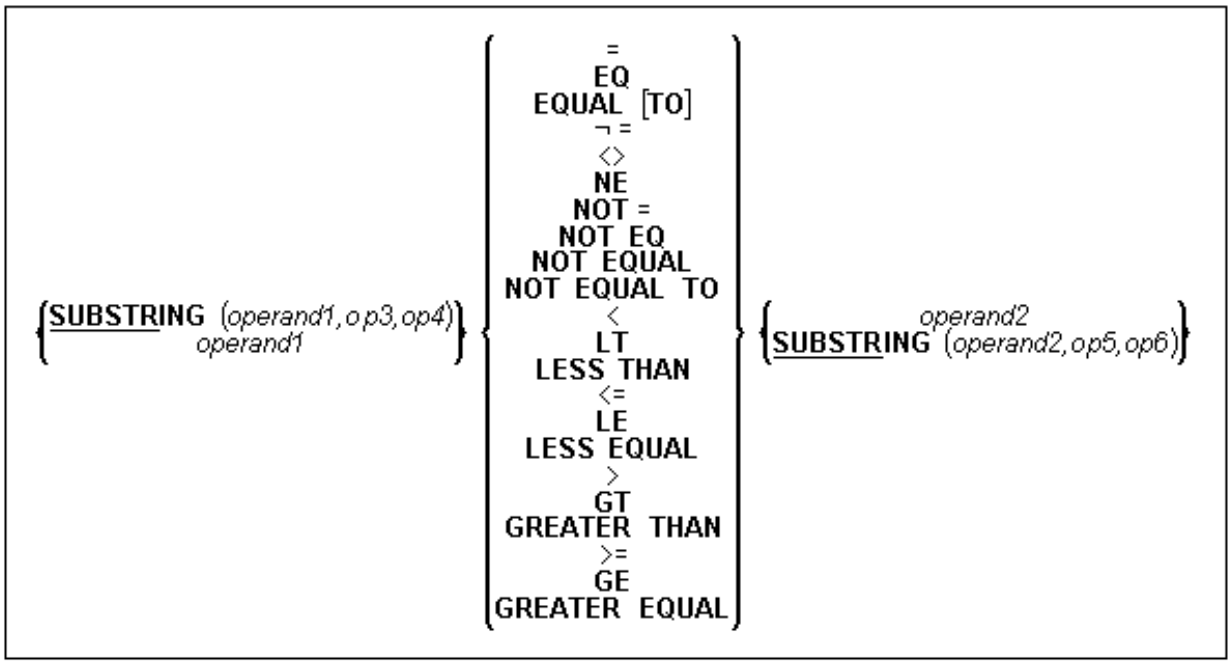
The following example shows how arithmetic expressions can be used in logical conditions:

```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

Handles in Logical Conditions

If the operands in a relation expression are handles, only EQUAL and NOT EQUAL operators may be used.

SUBSTRING Option in Relational Expression



Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A											yes	yes
Operand2	C	S	A		N	A											yes	no
Op3/Op5	C	S					N	P	I								yes	no
Op4/Op6	C	S					N	P	I								yes	no

With the SUBSTRING option, you can compare a *part* of an alphanumeric field. After the field name *(operand1)* you specify first the starting position *(op3)* and then the length *(op4)* of the field portion to be compared.

Also, you can compare a field value with part of another field value. After the field name *(operand2)* you specify first the starting position *(op5)* and then the length *(op6)* of the field portion *operand1* is to be compared with.

You can also combine both forms, that is, you can specify a SUBSTRING for both *operand1* and *operand2*.

Examples:

This expression compares the 5th to 12th position inclusive of the value in field #A with the value of field #B:

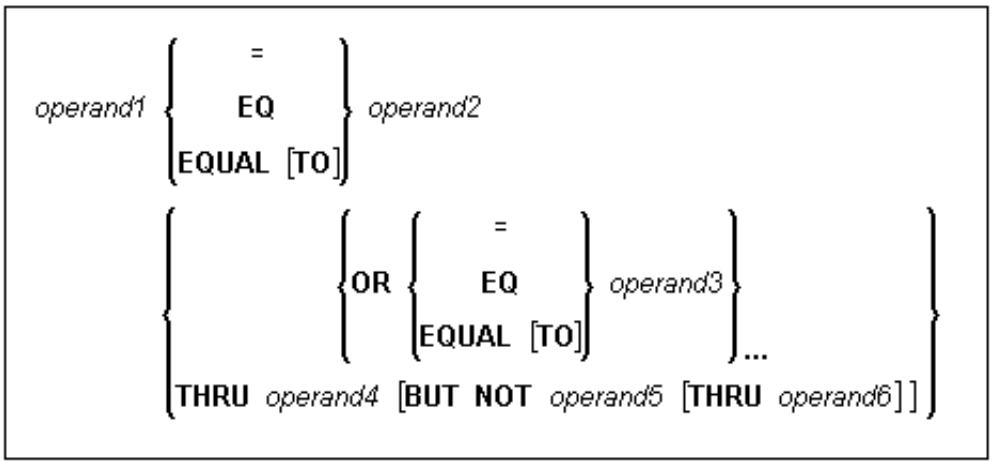
```
SUBSTRING(#A,5,8) = #B
```

This expression compares the value of field #A with the 3rd to 6th position inclusive of the value in field #B:

```
#A = SUBSTRING(#B,3,4)
```

Note:
If you omit op3/op5, the starting position is assumed to be "1". If you omit op4/op6, the length is assumed to be from the starting position to the end of the field.

Extended Relational Expression



Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no
Operand2	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no
Operand3	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no
Operand4	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no
Operand5	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no
Operand6	C	S	A		N* E	A	N	P	I	F	B	D	T			G	O	yes	no

* Mathematical functions and system variables are permitted.
Break functions are not permitted.

Operand3 can also be specified using a MASK or SCAN option; that is, it can be specified as:

MASK *(mask-definition) [operand]*
MASK *operand*
SCAN *operand*

For details on these options, see the sections MASK Option and SCAN Option.

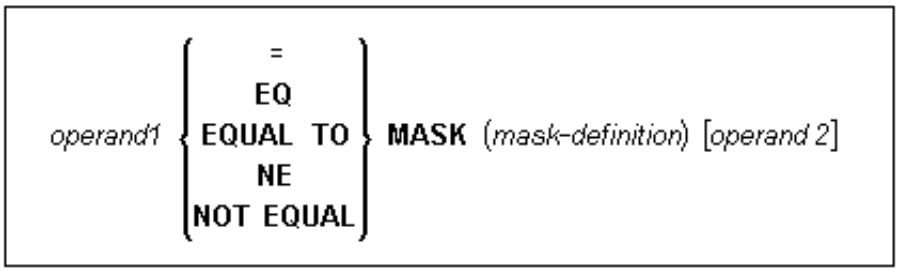
Examples:

```
IF #A = 2 OR = 4 OR = 7 IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

MASK Option

With the MASK option, you can check selected positions of a field for a specific content.

Constant Mask

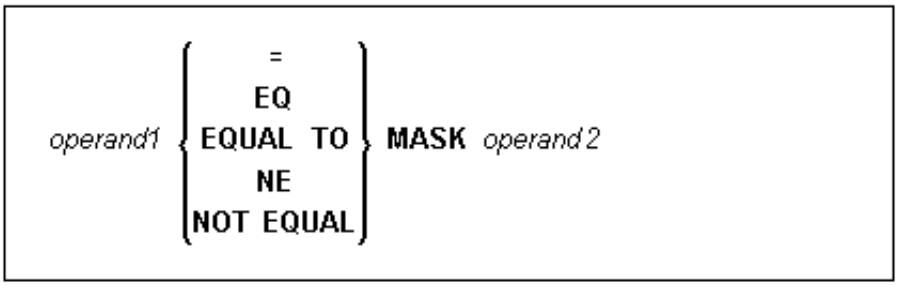


Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P									yes	no
Operand2	C	S				A	N	P			B						yes	no

Operand2 can only be used if the *mask-definition* contains at least one "X". *Operand1* and *operand2* must be format-compatible: if *operand1* is of format A, *operand2* must be of format A, B or N; if *operand1* is of format N or P, *operand2* must be of format N or P. An "X" in the the *mask-definition* selects the corresponding positions of the content of *operand1* and *operand2* for comparison.

Variable Mask

Apart from a constant *mask-definition* (see above), you may also specify a variable mask definition:



Operand	Possible Structure					Possible Formats										Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P								yes	no
Operand2		S				A										yes	no

The content of *operand2* will be taken as the mask definition. Trailing blanks in *operand2* will be ignored.

Characters in a Mask

The following characters may be used within a mask definition (the mask definition is contained in *mask-definition* for a constant mask and *operand2* for a variable mask):

Character	Meaning
. or ? or _	Indicates a single position that is not to be checked.
* or %	Indicates any number of positions not to be checked.
/	(Slash) Used to check if a value ends with a specific character (or string of characters). For example, the following condition will be true if there is either an "E" in the last position of the field, or the last "E" in the field is followed by nothing but blanks: IF #FIELD = MASK (*'E'/)
A	The position is to be checked for an alphabetical character (upper or lower case).
'*c*'	One or more positions are to be checked for the characters bounded by apostrophes (a double apostrophe indicates that a single apostrophe is the character to be checked for).
C	The position is to be checked for an alphabetical character (upper or lower case), a numeric character, or a blank.
DD	The two positions are to be checked for a valid day notation (01 - 31; dependent on the values of MM and YY/YYYY, if specified; see also Checking Dates).
H	The position is to be checked for hexadecimal content (A - F, 0 - 9).
L	The position is to be checked for a lower-case alphabetical character (a - z).
MM	The positions are to be checked for a valid month (01 - 12).
N	The position is to be checked for a numeric digit.
n...	One (or more) positions are to be checked for a numeric value in the range 0 - n.
n1-n2 or *n1:n2*	The positions are checked for a numeric value in the range n1-n2. n1 and n2 must be of the same length.
P	The position is to be checked for a displayable character (U, L, N or S).
S	The position is to be checked for special characters.
U	The position is to be checked for an upper-case alphabetical character (A - Z).
X	The position is to be checked against the equivalent position in the value (operand2) following the mask-definition. "X" is not allowed in a variable mask definition, as it makes no sense.
YY	The two positions are to be checked for a valid year (00 - 99). See also Checking Dates.
YYYY	The four positions are checked for a valid year (0000 - 2699).
Z	The position is to be checked for a character whose left half-byte is hexadecimally 3 or 7 (ASCII) or A - F (EBCDIC), and whose right half-byte is hexadecimally 0 - 9. This may be used to correctly check for numeric digits in negative numbers. With "N" (which indicates a position to be checked for a numeric digit), a check for numeric digits in negative numbers leads to incorrect results, because the sign of the number is stored in the last digit of the number, causing that digit to be hexadecimally represented as non-numeric. Within a mask, use only one "Z" for each sequence of numeric digits that is checked.

Mask Length

The length of the mask determines how many positions are to be checked.

Example:

```
DEFINE DATA LOCAL 1 #CODE (A15) END-DEFINE ... IF #CODE = MASK (NN'ABC'....NN) ...
```

The first two positions of #CODE are to be checked for numeric content. The three following positions are checked for the contents "ABC". The next four positions are not to be checked. Positions ten and eleven are to be checked for numeric content. Positions twelve to fifteen are not to be checked.

Checking Dates

Only one date may be checked within a given mask.

When dates are checked for a day (DD) and no month (MM) is specified in the mask, the current month will be assumed.

When dates are checked for a day (DD) and no year (YY or YYYY) is specified in the mask, the current year will be assumed.

When dates are checked for a 2-digit year (YY), the current century will be assumed if the profile parameter YSLW is set to "0". If the YSLW parameter is set to another value, the century will be determined by the "year sliding window" (as described under profile parameter YSLW in your Natural Operations documentation).

Examples:

Example 1:

```
MOVE 1131 TO #DATE (N4) IF #DATE = MASK (MMDD)
```

In this example, month and day are checked for validity. The value for month (11) will be considered valid, whereas the value for day (31) will be invalid since the 11th month has only 30 days.

Example 2:

```
IF #DATE(A8) = MASK (MM''DD''YY)
```

In this example, the content of the field #DATE is be checked for a valid date with the format MM/DD/YY (month/day/year).

Example 3:

```
IF #DATE (A4) = MASK (19-20YY)
```

In this example, the content of field #DATE is checked for a two-digit number in the range 19 to 20 followed by a valid two-digit year (00 through 99). The century is supplied by Natural as described above.
Note: Although apparent, the above mask does not allow to check for a valid year in the range 1900 through 2099, because the numeric value range 19-20 is checked independent of the year validation.
To check for year ranges, code one check for the date validation and another for the range validation:

```
IF #DATE (A10) = MASK (YYYYŸ-‡MMŸ-‡DD) AND #DATE = MASK (19-20)
```

Checking Against the Content of Constants or Variables

If the value for the mask check is to be taken from either a constant or a variable, this value *(operand2)* must be specified immediately following the *mask-definition*.

Operand2 must be at least as long as the mask.

In the mask, you indicate each position to be checked with "X", and each position not to be checked with "." (or "?" or "_").

Example:

```
DEFINE DATA LOCAL 1 #NAME (A15) END-DEFINE ... IF #NAME = MASK (..XX) 'ABCD' ...
```

It is checked whether the field #NAME contains "CD" in the third and fourth positions. Positions one and two are not checked.

The length of the mask determines how many positions are to be checked. The mask is left-justified against any field or constant used in the mask operation. The format of the field (or constant) on the right side of the expression must be the same as the format of the field on the left side of the expression.

If the field to be checked (operand1) is of format A, any constant used (operand2) must be enclosed in apostrophes. If the field is numeric, the value used must be a numeric constant or the content of a numeric database field or user-defined variable.

In either case, any characters/digits within the value specified which do not match positionally the "X" indicator within the mask are ignored.

The result of the MASK operation is true when the indicated positions in both values are identical.

Example:

```
/* EXAMPLE 'LCMSK0' /* EXAMPLE OF USING MASK OPTION WITHIN LOGICAL CONDITION /*************************** DEFINE DATA LOCAL 1 EMPLOY-VIEW VIEW OF EMPLOYEES  2 CITY END-DEFINE /*************************** HISTOGRAM EMPLOY-VIEW CITY  IF CITY = MASK (....XX) '....MN'  DISPLAY NOTITLE CITY 'NUMBER  END-IF END-HISTOGRAM /*************************** END
In the above example, the record will be accepted if the fifth and sixth positions of the field CITY each contain the character 'N'.
```

Range Checks

When performing range checks, the number of positions verified in the supplied variable is defined by the precision of the value supplied in the mask specification. For example, a mask of (...193...) will verify positions 4 to 6 for a three-digit number in the range 000 to 193.

Additional Examples of Mask Definitions:

- In this example, each character of #NAME is checked for an alphabetical character:
  ```
  IF #NAME (A10) = MASK (AAAAAAAAAA)
  ```
- In this example, positions 4 to 6 of #NUMBER are checked for a numeric value:
  ```
  IF #NUMBER (A6) = MASK (...NNN)
  ```
- In this example, positions 4 to 6 of #VALUE are to be checked for the value "123":
  ```
  IF #VALUE(A10) = MASK (...'123')
  ```
- This example will check if #LICENSE contains a license number which begins with "NY-" and whose last five characters are identical to the last five positions of #VALUE:
  ```
  DEFINE DATA LOCAL
     1 #VALUE(A8)
     1 #LICENSE(A8)
  END-DEFINE
  INPUT 'ENTER KNOWN POSITIONS OF LICENSE PLATE:' #VALUE
  ```



```
IF #LICENSE = MASK ('NY-'XXXXX) #VALUE
```

- The following condition would be met by any value which contains "NAT" and "AL" no matter which and how many other characters are between "NAT" and "AL" (this would include the values Natural and NATIONALITY as well as NATAL):
  `MASK('NAT'*'AL')`

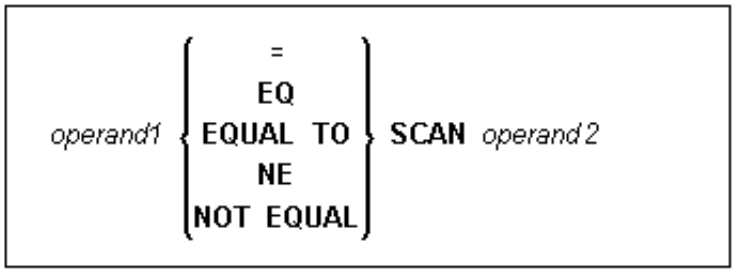
Checking Packed or Unpacked Numeric Data

Legacy applications often have packed or unpacked numeric variables redefined with alphanumeric or binary fields. Such redefinitions are not recommended, because using the packed or unpacked variable in an assignment or computation may lead to errors or unpredictable results. To validate the contents of such a redefined variable before the variable is used, use the N option as many as number of digits - 1 times followed by a single Z option.

Examples:

```
IF #P1 (P1) = MASK (Z)IF #N4 (N4) = MASK (NNNZ)IF #P5 (P5) = MASK (NNNNZ)
```

SCAN Option



Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P									yes	no
Operand2	C	S				A					B*						yes	no

* *Operand2* may only be binary if *operand1* is alphanumeric.

The SCAN option is used to scan for a specific value within a field.

The characters used in the SCAN option *(operand2)* may be specified as an alphanumeric constant (a character string bounded by apostrophes) or the contents of an alphanumeric database field or user-defined variable.

Trailing blanks are automatically eliminated from the value. Therefore, the SCAN option cannot be used to scan for blanks.

If *operand1* is alphanumeric, *operand2* may also be binary.

The field to be scanned *(operand1)* may be of format A, N or P. The SCAN operation may be specified with the equal (EQ) or not equal (NE) operators.

The length of the character string for the SCAN operation should be less than the length of the field to be scanned. If the length of the character string specified is identical to the length of the field to be scanned, then an EQUAL operator should be used instead of SCAN.

BREAK Within Logical Condition Criteria

```
BREAK [OF] operand1 [/n/]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A	N	P									yes	no

Note:
Dynamic or large variables are not allowed to be used as Operand1.

The BREAK option allows the current value or a portion of a value of a field to be compared to the value contained in the same field in the previous pass through the processing loop.

Operand1 specifies the field which is to be checked.

If only part of the field is to be checked, then specify a numeric constant *n*, enclosed by slashes, as the number of positions (counting from left to right) to be included in the comparison.

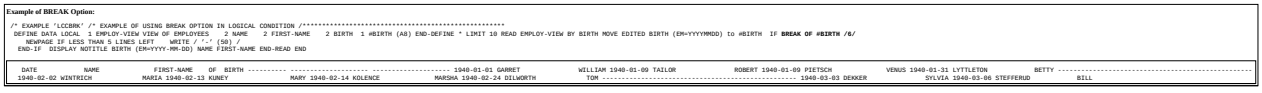
The result of the BREAK operation is true when a change in the specified positions of the field occurs. The result of the BREAK operation is not true if an AT END OF DATA condition occurs.

Example:

```
BREAK FIRST-NAME /1/
```

In this example, a check is made for a different value in the first position of the field FIRST-NAME.

Natural system functions (which are available with the AT BREAK statement) are *not* available with this option.



IS Option - Checking Format and Length of Value

```
operand1 IS (format)
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S	A			A											yes	no

This option is used to check whether the content of an alphanumeric field (operand1) can be converted to a specific other format.

This format for which the check is performed can be:

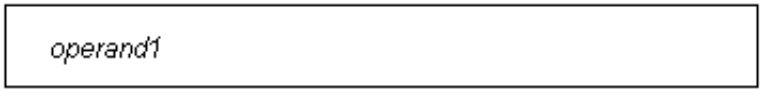
N*ll.ll*	Numeric with length *ll.ll.*
F*ll*	Floating point with length *ll.*
D	Date. The following date formats are possible: *dd-mm-yy*, *dd-mm-yyyy*, *ddmmyyyy* (*dd* = day, *mm* = month, *yy* or *yyyy* = year). The sequence of the day, month and year components as well as the characters between the components are determined by the profile parameter DTFORM (which is described in your Natural Operations documentation).
T	Time (according to the default time display format).
P*ll.ll*	Packed numeric with length *ll.ll*.
I*ll*	Integer with length *ll*.

When the check is performed, leading and trailing blanks in operand1 will be ignored.

The IS option may, for example, be used to check the content of a field before the mathematical function VAL (extract numeric value from an alphanumeric field) is used to ensure that it will not result in a runtime error.

Note:
The IS option cannot be used to check if the value of an alphanumeric field is in the specified "format", but if it can be **converted** to that "format". To check if a value is in a specific format, you can use the MASK option.

Evaluation of a Logical Variable



Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A											L				no	no

This option is used in conjunction with a logical variable (format L). A logical variable may take the value "TRUE" or "FALSE". As *operand1* you specify the name of the logical variable to be used.

Modified Control Variables

FIND	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S	A											C			no	no

This option is used to determine if the content of a field which has been assigned attributes dynamically has been modified during the execution of an INPUT statement.

Control variables referenced in an INPUT statement are always assigned the status "NOT MODIFIED" when the map is transmitted to the terminal.

Whenever the content of a field referencing a control variable is modified, the control variable is assigned the status "MODIFIED". When multiple fields reference the same control variable, the variable is marked "MODIFIED" if any of these fields is modified.

If *operand1* is an array, the result will be true if at least one of the array elements is modified (OR connection).

Note:
On mainframe computers, the profile parameter CVMIN (see the Natural Operations for Mainframes documentation) may be used to determine if a control variable is also to be set to "MODIFIED" if the value of the corresponding field is overwritten by an **identical** value.

SPECIFIED Option

This option is not available on mainframe computers.

This option is used to check whether an optional parameter in an invoked object (subprogram, external subroutine, dialog or ActiveX control) has received a value from the invoking object or not.

An optional parameter is a field defined with the keyword OPTIONAL in the DEFINE DATA PARAMETER statement of the invoked object. If a field is defined as OPTIONAL, a value can - but need not - be passed from an invoking object to this field.

In the invoking statement, the notation *n*X is used to indicate parameters for which no values are passed.

If you process an optional parameter which has not received a value, this will cause a runtime error. To avoid such an error, you use the SPECIFIED option in the invoked object to check whether an optional parameter has received a value or not, and then only process it if it has.

Parameter-name is the name of the parameter as specified in the DEFINE DATA PARAMETER statement of the invoked object.

For a field not defined as OPTIONAL, the SPECIFIED condition is always "TRUE".

Fields Used Within Logical Condition Criteria

Database fields and user-defined variables may be used to construct logical condition criteria. A database field which is a multiple-value field or is contained in a periodic group can also be used. If a range of values for a multiple-value field or a range of occurrences for a periodic group is specified, the condition is true if the search value is found in any value/occurrence within the specified range.

Each value used must be compatible with the field used on the opposite side of the expression. Decimal notation may be specified only for values used with numeric fields, and the number of decimal positions of the value must agree with the number of decimal positions defined for the field.

If the operands are not of the same format, the second operand is converted to the format of the first operand.

The following table shows which operand formats can be used together in a logical condition:

Operand2 → ↓ Operand1	**A**	**B*n* (*n*≤4)**	**B*n* (*n*≥5)**	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
A	Y	Y	Y									
B*n* (*n*≤4)	Y	Y	Y			Y	Y		Y	Y		
B*n* (*n*≥5)	Y	Y	Y									
D				Y								
T					Y							
I		Y				Y	Y		Y	Y		
F		Y				Y	Y		Y	Y		
L												
N		Y				Y	Y		Y	Y		
P		Y				Y	Y		Y	Y		
GH											Y	
OH												Y

GH = GUI handle, OH = object handle.

If an array is compared with a scalar value, each element of the array will be compared with the scalar value. The condition will be true if at least one of the array elements meets the condition (OR connection).

If an array is compared with an array, each element in the array is compared with the corresponding element of the other array. The result is true only if all element comparisons meet the condition (AND connection).

See also Processing of Arrays.

An Adabas phonetic descriptor cannot be used within a logical condition.

Examples of Logical Condition Criteria:

```
FIND EMPLOYEES-VIEW WITH CITY = 'BOSTON' WHERE SEX = 'M' READ EMPLOYEES-VIEW BY NAME WHERE SEX = 'M' ACCEPT IF LEAVE-DUE GT 45 IF #A GT #B THEN COMPUTE #C = #A + #B REPEAT UNTIL #X = 500
```

Logical Operators in Complex Logical Expressions

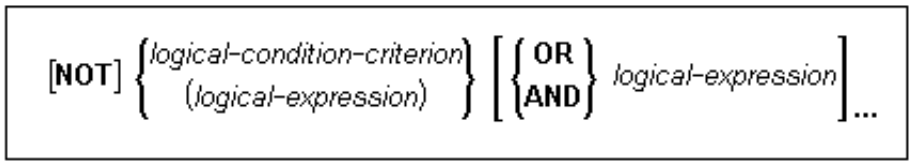
Logical condition criteria may be combined using the Boolean operators "AND", "OR", and "NOT". Parentheses may also be used to indicate logical grouping.

The operators are evaluated in the following order:

1. **( )** Parentheses
2. **NOT** Negation
3. **AND** AND connection
4. **OR** OR connection

The following *logical-condition-criteria* may be connected by logical operators to form a complex *logical-expression*: relational expressions, extended relational expressions, MASK, SCAN, and BREAK options.

The syntax for a *logical-expression* is as follows:



Examples of Logical Expressions:

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'              WHERE BIRTH GT 19610101 AND SEX = 'F'IF NOT (#CITY = 'A' THRU 'E')
```

For information on comparing arrays in a logical expression, see Processing of Arrays.

Note:
If multiple logical-condition-criteria are connected with "AND", the evaluation terminates as soon as the first of these criteria is not true.

Rules for Arithmetic Assignment

- Field Initialization
- Data Transfer
- Field Truncation and Field Rounding
- Result Format and Length in Arithmetic Operations
- Arithmetic Operations with Floating-Point Numbers
- Arithmetic Operations with Date and Time
- Performance Considerations for Mixed Format Expressions
- Precision of Results for Arithmetic Operations
- Error Conditions in Arithmetic Operations
- Processing of Arrays

Field Initialization

A field - user-defined variable or database field - which is to be used as an operand in an arithmetic operation must be defined with one of the following formats: N, P, I, F, D, T.

Note for reporting mode:
A field which is to be used as an operand in an arithmetic operation must have been previously defined.
A user-defined variable or database field used as a result field in an arithmetic operation need not have been previously defined.

All user-defined variables and all database fields defined in a DEFINE DATA statement are initialized to the appropriate zero or blank value when the program is invoked for execution.

Data Transfer

Data transfer is performed with a MOVE or COMPUTE statement. The following table summarizes data transfer compatibility and the rules for data transfer:

Sending Field	Receiving Field											
	N / P	**A**	**B*n*** **(*n* < 5)**	**B*n*** **(*n* > 4)**	**I**	**L**	**C**	**D**	**T**	**F**	**GH**	**OH**
N or **P**	Y	[2]	[3]	-	Y	-	-	-	Y	Y	-	-
A	-	Y	[1]	[1]	-	-	-	-	-	-	-	-
B*n* **(*n* < 5)**	[4]	[2]	[5]	[5]	Y	-	-	-	Y	Y	-	-
B*n* **(*n* > 4)**	-	[6]	[5]	[5]	-	-	-	-	-	-	-	-
I	Y	[2]	[3]	-	Y	-	-	-	Y	Y	-	-
L	-	[9]	-	-	-	Y	-	-	-	-	-	-
C	-	-	-	-	-	-	Y	-	-	-	-	-
D	Y	[9]	Y	-	Y	-	-	Y	[7]	Y	-	-
T	Y	[9]	Y	-	Y	-	-	[8]	Y	Y	-	-
F	Y	[9][10]	[3]	-	Y	-	-	-	Y	Y	-	-
GH	-	-	-	-	-	-	-	-	-	-	Y	-
OH	-	-	-	-	-	-	-	-	-	-	-	Y

Y	indicates data transfer compatibility.
-	indicates data transfer incompatibility.
[1]...	refers to the data conversion rules.

GH = GUI handle, OH = object handle.

See also Usage of Dynamic Variables.

Data Conversion

The following rules apply to converting data values:

1. **Alphanumeric to binary:** The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blank characters depending on the length defined and the number of bytes specified.
2. **(N,P,I) and binary (length 1-4) to alphanumeric:** The value will be converted to unpacked form and moved into the alphanumeric field left justified, i.e., leading zeros will be suppressed and the field will be filled with trailing blank characters. For negative numeric values, the sign will be converted to the hexadecimal notation "Dx". Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.
3. **(N,P,I,F) to binary (1-4 bytes):** The numeric value will be converted to binary (4 bytes). Any decimal point in the numeric value will be ignored (the digits of the value before and after the decimal point will be treated as an integer value). The resulting binary number will be positive or a two’s complement of the number depending on the sign of the value.
4. **Binary (1-4 bytes) to numeric:** The value will be converted and assigned to the numeric value right justified, i.e., with leading zeros. (Binary values of the length 1-3 bytes are always assumed to have a positive sign. For binary values of 4 bytes, the leftmost bit determines the sign of the number: 1=negative, 0=positive.) Any decimal point in the receiving numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.

5. **Binary to binary:** The value will be moved from right to left byte by byte. Leading binary zeros will be inserted into the receiving field.
6. **Binary (>4 bytes) to alphanumeric:** The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks depending on the length defined and the number of bytes specified.
7. **Date (D) to time (T):** If date is moved to time, it is converted to time assuming time 00:00:00:0.
8. **Time (T) to date (D):** If time is moved to date, the time information is truncated, leaving only the date information.
9. **L,D,T,F to A:** The values are converted to display form and are assigned left justified.
10. If F is assigned to an alphanumeric field which is too short, the mantissa is reduced accordingly.

See also Usage of Large and Dynamic Variables/Fields.

Field Truncation and Field Rounding

The following rules apply to field truncation and rounding:

- High-order numeric field truncation is allowed only when the digits to be truncated are leading zeros. Digits following an expressed or implied decimal point may be truncated.
- Trailing positions of an alphanumeric field may be truncated.
- If the option ROUNDED is specified, the last position of the result will be rounded up if the first truncated decimal position of the value being assigned contains a value greater than or equal to 5.
  For the result precision of a division, see also Precision of Results for Arithmetic Operations.

Result Format and Length in Arithmetic Operations

The following table shows the format and length of the result of an arithmetic operation:

	I1	**I2**	**I4**	**N or P**	**F4**	**F8**
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N or P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

P* is determined from the integer length and precision of the operands individually for each operation, as shown under Precision of Results for Arithmetic Operations.

The following decimal integer lengths and possible values are applicable for format I:

Format/Length	**Decimal Integer Length**	**Possible Values**
I1	3	-128 to 127
I2	5	-32 768 to 32 767
I4	10	-2 147 483 648 to 2 147 483 647

Arithmetic Operations with Floating-Point Numbers

Some General Considerations

Floating-point numbers (format F) are represented as a sum of powers of two (as are integer numbers (format I)), whereas unpacked and packed numbers (formats N and P) are represented as a sum of powers of ten.

In unpacked or packed numbers, the position of the decimal point is fixed. In floating-point numbers, however, the position of the decimal point (as the name indicates) is "floating", that is, its position is not fixed, but depends on the actual value.

Floating-point numbers are essential for the computing of trigonometric functions or mathematical functions such as sinus or logarithm.

The Precision of Floating-Point Numbers

Due to the nature of floating-point numbers, their precision is limited:

- For a variable of format/length F4, the precision is limited to approximately 7 digits.
- For a variable of format/length F8, the precision is limited to approximately 15 digits (16 digits on mainframe computers).

Values which have more significant digits cannot be represented exactly as a floating-point number. No matter how many additional digits there are before or after the decimal point, a floating-point number can cover only the leading 7 or 15 (16) digits respectively.

An integer value can only be represented exactly in a variable of format/length F4 if its absolute value does not exceed 2^{23} -1 (2^{24} -1 on mainframe computers).

Conversion to Floating-Point Representation

When an alphanumeric, unpacked numeric or packed numeric value is converted to floating-point format (for example, in an assignment operation), the representation has to be changed, that is, a sum of powers of ten has to be converted to a sum of powers of two.

Consequently, only numbers that are representable as a finite sum of powers of two can be represented exactly; all other numbers can only be represented approximately.

Examples:

This number has an exact floating-point representation:

$1.25 = 2^{0} + 2^{-2}$

This number is a periodic floating-point number without an exact representation:

$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$

Thus, the conversion of alphanumeric, unpacked numeric or packed numeric values to floating-point values, and vice versa, can introduce small errors.

Platform-Dependent Differences

As already indicated by some of the differing limits mentioned above, the representation of floating-point numbers on mainframe computers is different from their representation on other platforms.

This explains why the same application, when run on different platforms, may return slightly different results when floating-point arithmetics are involved.

If you port a Natural application to another platform, also remember that the range of possible values for floating-point variables on a mainframe computer is different from that on other platforms:

- The possible value range for F4 and F8 variables on a mainframe computer is (approximately):
  $\pm 5.4 * 10^{-79}$ to $\pm 7.2 * 10^{75}$
- The possible value range on most other platforms is (approximately):
  for F4 variables: $\pm 1.17 * 10^{-38}$ to $\pm 3.40 * 10^{38}$
  for F8 variables: $\pm 2.22 * 10^{-308}$ to $\pm 1.79 * 10^{308}$

Note:
The representation used by your pocket calculator may also be different from the one used by your computer - which explains why results for the same computation may differ.

Arithmetic Operations with Date and Time

With formats D (date) and T (time), only addition and subtraction are allowed; multiplication and division are not allowed.

Date/time values can be added to/subtracted from one another; or integer values (no decimal digits) can be added to/subtracted from date/time values. Such integer values can be contained in fields of formats N, P, I, D, or T.

An integer value added to/subtracted from a date value is assumed to be in days. An integer value added to/subtracted from a time value is assumed to be in tenths of seconds.

For arithmetic operations with date and time, certain restrictions apply, which are due to the Natural's internal handling of arithmetic operations with date and time, as explained below.

Internally, Natural handles an arithmetic operation with date/time variables as follows:

COMPUTE *result-field = operand1 +/- operand2*

The above statement is resolved as:

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

That is, in a first step Natural computes the result of the addition/subtraction, and in a second step assigns this result to the result field.

More complex arithmetic operations are resolved following the same pattern:

COMPUTE *result-field = operand1 +/- operand2 +/- operand3 +/- operand4*

The above statement is resolved as:

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

The internal format of such an *intermediate-result* depends on the formats of the operands, as shown in the tables below.

The following table shows the format of the *intermediate-result* of an **addition** (*intermediate-result = operand1 + operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	**D**	**D***i*
D	**T**	**T**
D	**N, P, I**	**D**
T	**D, T, N, P, I**	**T**
N, P, I	**D**	**D**
N, P, I	**T**	**T**

The following table shows the format of the *intermediate-result* of a **subtraction** (*intermediate-result = operand1 - operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	**D**	**D***i*
D	**T**	**T***i*
D	**N, P, I**	**D**
T	**D, T**	**T***i*
T	**N, P, I**	**T**
N, P, I	**D**	**D***i*
N, P, I	**T**	**T***i*

D*i* is a value in internal date format; **T***i* is a value in internal time format; such values can be used in further arithmetic date/time operations, but they cannot be assigned to a result field of format D (see the assignment table below).

In complex arithmetic operations in which an intermediate result of internal format **D***i* or **T***i* is used as operand in a further addition/subtraction, its format is assumed to be D or T respectively.

The following table shows which intermediate results can internally be assigned to which result fields (*result-field = intermediate-result*).

Format of *result-field*	Format of *intermediate-result*	Assignment possible
D	**D**, **T**	yes
D	**D***i*, **T***i*, **N**, **P**, **I**	*no*
T	**D**, **T**, **D***i*, **T***i*, **N**, **P**, **I**	yes
N, **P**, **I**	**D**, **T**, **D***i*, **T***i*, **N**, **P**, **I**	yes

A result field of format D or T must not contain a negative value.

Examples 1 and 2 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format D*i*, and a value of format D*i* cannot be assigned to a result field of format D.

Examples 3 and 4 (invalid):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T) COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format T*i*, and a value of format T*i* cannot be assigned to a result field of format D.

Example 5 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

This operation is possible. First, DATE3 is subtracted from DATE2, giving an intermediate result of format D*i*; then, this intermediate result is added to TIME3, giving an intermediate result of format T; finally, this second intermediate result is assigned to the result field DATE1.

If a format T value is assigned to a format D field, you must ensure that the time value contains a valid date component.

Performance Considerations for Mixed Format Expressions

When doing arithmetic operations, the choice of field formats has considerable impact on performance:

For business arithmetic under OpenVMS, UNIX and Windows, only fields of format I (integer) should be used.
If your computer is equipped with a math co-processor, format F (floating point) is faster than formats N or P and almost as fast as format I.
Without a math co-processor, format F is approximately as slow as formats N or P.

For business arithmetic on mainframe computers, only fields of format P (packed numeric) should be used. The number of decimal digits in all operands should agree where possible.

For scientific arithmetic, only fields of format F (floating point) should be used.

In expressions where formats are mixed between numeric (N, P) and floating point (F), a conversion to floating point format is performed. This conversion results in considerable CPU load. Therefore it is recommended to avoid mixed format expressions in arithmetic operations.

Precision of Results for Arithmetic Operations

Operation	Digits Before Decimal Point	Digits After Decimal Point
Addition/Subtraction	**Fi + 1** or **Si + 1** (whichever is greater)	**Fd** or **Sd** (whichever is greater)
Multiplication	**Fi + Si + 2**	**Fd + Sd** (maximum 7)
Division	**Fi + Sd**	(see below)
Exponentiation	**15 - Fd** (Exception: On mainframe computers, if the exponent has one or more digits after the decimal point, and on all other platforms in general, the exponentiation is internally carried out in floating point format. See Arithmetic Operations with Floating-Point Numbers for further information.)	**Fd**
Square Root	**Fi**	**Fd**
F = First operand **S** = Second operand **R** = Result **i** = Digits before decimal point **d** = Digits after decimal point		

Digits after Decimal Point for Division Results

The precision of the result of a division depends whether a result field is available or not:

- If a result field is available, the precision is: **Rd** or **Fd** (whichever is greater) *.
- If no result field is available, the precision is: **Fd** or **Sd** (whichever is greater) *.

A result field *is* available (or assumed to be available) in a COMPUTE and DIVIDE statement, and in a logical condition in which the division is placed after the comparison operator (for example: IF #A = #B / #C THEN ...).
A result field is not (or not assumed to be) available in a logical condition in which the division is placed before the comparison operator (for example: IF #B / #C = #A THEN ...).

Exception: If both dividend and divisor are of integer format and at least one of them is a variable, the division result is always of integer format (regardless of the precision of the result field and of whether the ROUNDED option is used or not).

* If the ROUNDED option is used, the precision of the result is internally increased by one digit before the result is actually rounded.

Error Conditions in Arithmetic Operations

In an addition, subtraction, multiplication or division, an error occurs if the total number of digits (before and after the decimal point) of the result is greater than 31.

In an exponentiation, an error occurs in any of the following situations:

- if the base is of packed format and either the result has over 16 digits or any intermediate result has over 15 digits;
- if the base is of floating-point format and the result is greater than approximately $7 * 10^{75}$.

Processing of Arrays

Generally, the following rules apply:

- All scalar operations may be applied to array elements which consist of a single occurrence.
- If a variable is defined with a constant value (for example, #FIELD (I2) CONSTANT <8>), the value will be assigned to the variable at compilation, and the variable will be treated as a constant. This means that if such a variable is used in an array index, the dimension concerned has a *definite* number of occurrences.
- If an assignment/comparison operation involves two arrays with a different number of dimensions, the "missing" dimension in the array with fewer dimensions is assumed to be (1:1).

Example:

If **#ARRAY1 (1:2)** is assigned to **#ARRAY2 (1:2,1:2)**,
#ARRAY1 is assumed to be **#ARRAY1 (1:1,1:2)**.

Assignment Operations with Arrays

If an array range is assigned to another array range, the assignment is performed element by element.

If a single occurrence is assigned to an array range, each element of the range is filled with the value of the single occurrence. (For a mathematical function, each element of the range is filled with the result of the function.)

Before an assignment operation is executed, the individual dimensions of the arrays involved are compared with one another to check if they meet one of the conditions listed below. The dimensions are compared independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The assignment of values from one array to another is only allowed under one of the following conditions:

- The number of occurrences is the same for both dimensions compared.
- The number of occurrences is indefinite for both dimensions compared.
- The dimension that is assigned to another dimension consists of a single occurrence.

The following program shows which array assignment operations are possible.

Example - Array Assignments:

Comparison Operations with Arrays

Generally, the following applies: if arrays with multiple dimensions are compared, the individual dimensions are handled independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The comparison of two array dimensions is only allowed under one of the following conditions:

- The array dimensions compared with one another have the same number of occurrences.
- The array dimensions compared with one another have an indefinite number of occurrences.
- All array dimensions of one of the arrays involved are single occurrences.

The following program shows which array comparison operations are possible:

Example - Array Comparisons:

When you compare two array ranges, note that the following two expressions lead to different results:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
 NOT #ARRAY1(*) = #ARRAY2(*)
```

Example:

Condition A:

```
IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
```

This is equivalent to:

```
IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
```

Condition A is therefore true if the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *and* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

Condition B:

```
IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
```

This is equivalent to:

```
IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
```

This in turn is equivalent to:

```
IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
```

Condition B is therefore true if *either* the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *or* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

Arithmetic Operations with Arrays

In arithmetic operations (in COMPUTE, ADD or MULTIPLY statements), array ranges may be used in the following ways:

- *range* + *range* = *range*.
  The range dimensions must be equal.
  The addition is performed element by element.
- *range* * *range* = *range*.
  The range dimensions must be equal.
  The multiplication is performed element by element.
- *scalar* + *range* = *range*.
  The range dimensions must be equal.
  The scalar is added to each element of the range.
- *range* * *scalar* = *range*.
  The range dimensions must be equal.
  Each element of the range is multiplied by the scalar.
- *range* + *scalar* = *scalar*.
  Each element of the range is added to the scalar and the result is assigned to the scalar.
- *scalar* * *range* = *scalar2*.
  The scalar is multiplied by each element of the array and the result is assigned to scalar2.

Renumbering of Source-Code Line Number References

Numeric four-digit source-code line numbers that reference a statement (see Statement Reference Notation - r) are also renumbered if the Natural source program is renumbered. For the user's convenience and to aid in readability and debugging, all source code line number references that occur in a statement, an alphanumeric constant or a comment are renumbered. The position of the source code line number reference in the statement or alphanumeric constant (start, middle, end) does not matter.

The following patterns are recognized as being a valid source code line number reference and are renumbered (*nnnn* is a four-digit number):

Pattern	Sample Statement
(nnnn)	`ESCAPE BOTTOM (0150)`
(nnnn /	`DISPLAY ADDRESS-LINE(0010/1:5)`
(nnnn,	`DISPLAY NAME(0010,A10/1:5)`

If the left parenthesis or the four-digit number nnnn is followed by a blank, or the four-digit number *nnnn* is followed by a period, the pattern is not considered to be a valid source code line number reference.

To avoid that a four-digit number that is contained in an alphanumeric constant is unintentionally renumbered, the constant should be split up and the different parts should be concatenated to form a single value by use of a hyphen.

Example:
`Z := 'XXXX (1234,00) YYYY'` should be replaced by
`Z := 'XXXX (1234' - ',00) YYYY'`

